Algorithmic Design: Cuckoo Hashing

Bruno Bonaiuto Bolivar

Course of AA 2022-2023 - DSSC

1 Introduction

A dictionary is a fundamental data structure in computer science, allowing us to store key-value pairs, and also to lookup by key the corresponding value in data structure. The hash table provides one of the most common and one of the fastest implementations of the dictionary data structure, allowing for amortized constant-time lookup, insertion and deletion operations. In a typical from, a hash function h computed on the space of keys is used to map each key to a location in a set of bucket locations $\{b_0, ..., b_{r-1}\}$. The *key* and some auxiliary data 'value' is stored at the appropriate bucket location in 'Insertion'. Then for deletion, the key with its value is removed from the bucket. And in 'Lookup' the key-value pair, if it exits, ise retrieved by looking at the bucket location corresponding to the hash key-value.

In hash tables, hash functions on keys are used to map possible keys to buckets in such a way that it is unlikely that too distinct keys are mapped to the same bucket. However, in the absence of perfect hash functions mapping keys injectively to locations in a set of buckets, collisions where two distinct keys map to a single bucket location must be resolved. There are several Schemes to resolver collisions with different performance characteristics, cuckoo hashing is a simple one for solving this collisions allowing constant-time worst-case lookup and deletion operations, and also amortized constant-time insertion operations.

2 Previous hashing schemes

Chained Hashing a **linked list** is used to store all keys hashing to a given location in the set of buckets. Collision are resolved by lengthening the list structures in buckets with collisions. **insertion operations** can be done in constant time by appending or prepending to the list for the relevant bucket, by **lookup and deletion operation** may require traversal of the entire list. performance of the hash table then degrades when theses list grow large.**this is Optional** *for example: when the load factor for the hash table is high. by choosing appropriate times to resize the hash table, lookup and deletion can be perform in amortized constant time. but in particular, unlike cuckoo hashing, there is not constant-time worst-case lookup guarantee.*

Linear Probing (the next free slot) In a linear probing scheme, keys are hashed to form a bucket index and if possible, are stored in that bucket location. On insertion, if the bucket location is already taken by another key, the bucket locations are scanned in order to find a free bucket to store the key. During lookup for a certain key, that key is hashed to provide a bucket location, and consecutive buckets starting with that bucket are scanned until the key is found or an empty bucket is encountered. Certainly, these insertion and lookup schemes do not have reasonable constant-time worst-case run-time bounds, since large clusters of contiguous non- empty buckets may force many buckets to be accessed before a correct bucket is found.

Linear probing is a type of open addressing collision resolution scheme, where a hash collision is resolved by probing through a set of alternate locations in an array of buckets. Similar open addressing schemes include quadratic probing, where the interval between probes increases quadratically, and double hashing, where the distance between probes for a given key is linear, but is given by the value of another hash function on that key.

Two-way chaining two hash tables with two independent hash functions are used so that each possible key hashes to one bucket location in each hash table. Each bucket holds a list of items that hash to that bucket, as in the case of chained hashing. When a key is inserted, it is inserted into the table whose corresponding bucket location has the list with the fewest number of keys. During lookup and deletion, lists in both tables are traversed to locate a given key. Performance here can be better than in the chained hashing case because the action of choosing the bucket with the fewest number of existing keys serves to, with high probability, even out the distribution of keys to buckets. This scheme shares with Cuckoo Hashing its idea of designating possible locations in two hash tables to store a given key.

Up to here all the Good

3 cuckoo hashing

(doesnt use perfect hashing, instead it use a variant of open adressing) Is a hash table scheme using two hash tables T_1 and T_2 each with r buckets with independent hash functions h_1 and h_2 each mapping a universe U to bucket locations $\{0, ..., r-1\}$. Respectively, A key x can be stored in exactly one of the locations $T_1[h_1(x)]$ and $T_2[h_2(x)]$ but never in both. **The lookup operation** in Cuckoo Hashing examines both locations $T_1[h_1(x)]$ and $T_2[h_2(x)]$ and succeeds if the key x is stored in either location. Formally, the following algorithm describes the lookup behaviour of Cuckoo Hashing.

Pseudo-code of Lookup here

Certainly, this algorithm performs in worst-case constant time. **Deletion is also a simple operation**, since we may simply remove a key from its containing bucket.

3.1 Two equal keys

Given hash functions h_1 and h_2 and a set of keys to be stored in our hash tables, how likely is it that these keys be placed into buckets so that for each key x, either $T_1[h_1(x)] = x$ and $T_2[h_2(x)] = x$; if the tables are a bit less than half full, and h_1 and h_2 are picked uniformly randomly, if $r \geq (1+\epsilon)n$, for example: each of the two hash tables has size at least $1 + \epsilon$ times the total number of keys, for some constant $\epsilon > 0$, the probability that the keys cannot be put in such a configuration is $O(1/n)$.

3.2 Insertion

Resumed More constructively, we demonstrate a simple insertion procedure that allows for amortized constant-time insertion operations. The insertion procedure, we place a key in one of its designated places in the two tables, evicting any key already present. We subsequently try to insert the evicted key into its alternative bucket location in the other hash table, potentially evicting another key, and continuing in this way until a key is inserted into an empty bucket, or after a number of iterations, we decide to rehash the table and try again.

for the presentation As for insertion, it turns out that the "cuckoo approach", kicking other keys away until every key has its own "nest", works very well. Specifically, if x is to be inserted we first see if cell $h_1(x)$ of T_1 is occupied. If not, we are done. Otherwise we set $T_1[h_1(x)] \leftarrow x$ anyway, thus making the previous occupant "nestless." This key is then inserted in T_2 in the same way, and so forth attractively

Loops In that way it may happen that this process loops, as shown in Fig. 1(b). Therefore the number of iterations is bounded by a value "MaxLoop" to be specified in Section 2.3. If this number of iterations is reached, we rehash the keys in the tables using new hash functions, and try once again to accommodate the nestless key. There is no need to allocate new tables for the rehashing: We may simply run through the tables to delete and perform the usual insertion procedure on all keys found not to be at their intended position in the table.

Pseudo-code of Insertion here

The procedure insert assumes that each table remains larger than $(1 + \epsilon)n$ cells. When no such bound is known, a test must be done to find out when a rehash to larger tables is needed. If the hash tables have size r, we enforce that no more than r^2 insertions are performed without changing the hash functions. More specifically, if r^2 insertions have been performed since the beginning of the last rehash, we force a new rehash.

As with many other hashing schemes, **performance deteriorates as the load factor** r/n **increases.** We therefore maintain that our hash tables each have size at least $r \geq (1+\epsilon)n$ for some constant ϵ. We do this by doubling the size of the table and rehashing the table whenever $r < (1+\epsilon)n$ .The amortized

```
procedure insert(x)
    if lookup(x) then return
    loop MaxLoop times
        $x \leftrightarrow T_1[h_1(x)]$
        if $x = \perp$ then return
        $x \leftrightarrow T_2[h_2(x)]$
        if $x = \perp$ then return
    end loop
    rehash(); insert(x);
end
```

Figure 1: lookup

cost of **resizing** the hash table per insertion is $O(1)$. And **rehashing** a table of size n requires $O(n)$ expected time.

3.3 Hash family assumption Optional

Considering the probability a set of keys will hash to a certain arrangement of buckets, we need a universality guarantee on the hash family from which h_1 and h_2 are selected. Pagh and Rodler develop the notion of (c, k)-universal hash functions to provide this guarantee. For simplified analysis, **we make the assumption that with probability $1 - O(1/n^2)$ our hash function h_1 and h_2 are both drawn randomly from all mappings $O \to 0, ..., r - 1$, and with probability $O(1/n^2)$, we make no assumptions about h**

4 Comparison to previous hash families

Pagh and Rodler [4] performed experiments measuring the run-time of Cuckoo Hashing with optimized implementations of chained hashing, linear probing and two-way chaining. On a clock cycle basis, Cuckoo Hashing is quite competitive. We have reproduced the results of these experiments in Figure 7, where various operations were performed with a load factor of 1/3. Note that for values of n below 215, run-times of the various hash table schemes are quite similar, due to the effect of CPU caching. With values of n above that threshold, it is likely that cache miss effects explain the difference in run-time of the various schemes.

5 insertion overview optional, explaining the cases of insertion